Improving Product Configuration in Software Product Line Engineering

Lei Tan **Yuqing Lin** **Huilin Ye** **Guoheng Zhang**

School of Electrical Engineering and Computer Science
University of Newcastle,
University Drive, Callaghan, NSW, 2308,
Email: {lei.tan, guoheng.zhang}@uon.edu.au,
{yuqing.lin, huilin.ye}@newcastle.edu.au

Abstract

Software Product Line Engineering (SPLE) is a emerging software reuse paradigm. SPLE focuses on systematic software reuse from requirement engineering to product derivation throughout the software development life-cycle. Feature model is one of the most important reusable assets which represents all design considerations of a software product line. Feature model will be used in the product configuration process to produce a software. The product configuration is a decision-making process, where all kinds of relationships among configurable features will be considered to select the desired features for the product. To improve the efficiency and quality of product configuration, we are proposing a new approach which aims at identifying a small set of key features. The product configuration should always start from this set of features since, based on the feature dependencies, the decisions made on these features will imply decisions on the rest of the features of the product line, thus reduce the features visited in the configuration process. We have also conducted some experiments to demonstrate how the proposed approach works and evaluate the efficiency of the approach.

Keywords: Software Product Line; Feature Model; Product Configuration; Minimum Vertex Cover.

1 Introduction

During the last decade, software product line (SPL) engineering has emerged as an effective software development methodology to promote systematic software reuse. An SPL is a collection of software products that share common characteristics as a family. The key idea of software product line engineering is to discover and exploit commonalities across a product family, thus to improve the reusability of various software engineering assets. A successful SPL based software development will improve the development productivity and the quality of software, and significantly reduce development cost and time-to-market.

This work is supported by Australian Research Council under Discovery Project DP0772799.

In addition to the commonalities shared by all the products in an SPL, individual products may somehow vary from each other. The variabilities among products in an SPL must be appropriately represented and managed. Feature oriented modelling approaches have been widely used in software product line engineering for this purpose. Features are prominent and distinctive system requirements or characteristics that are visible to various stakeholders in a product line (Lee et al. 2002). A feature model specifies the features, their relationships, and the constraints of feature selections for product configuration. A product in an SPL is defined by a unique valid combination of selected features. Product configuration is a process of selecting features for developing a product in an SPL.

A feature model is usually represented as a tree in which the variabilities of features are represented as variation points (VPs). A variation point (Pohl et al. 2005) consists of a parent feature, a group of child features, called variants, and a multiplicity specifying the minimum and maximum number of variants that can be selected from the variation point when configuring a product. The selection of variants at a variation point is not only constrained by the multiplicity but also by the dependencies between the variants at this variation point and the variants at other variation points. Dependencies are the constraints on configurations in a product line. The following two dependencies have been defined by Kang et al. (1990).

1. Requires: If a feature requires, or uses, another feature to fulfil its task, there is a Requires relationship between these two features.
2. Excludes: If a feature has conflicts with another feature, they cannot be chosen for the same product configuration, i.e. they mutually exclude each other. There is a bi-directional Excludes relationship between two features.

Many other types of dependencies have been considered as well, such as *Impact*, *Mandatory*, *Optional*, *Alternative* and *Or* (Benavides et al. 2010, Ye et al. 2008). It does not seem that there is an agreeable industry standard for the types of dependencies to be included in the feature model.

A valid feature model describes the configuration space of a system family (Czarnecki et al. 2005). During product configuration process, application engineers specify member products by selecting the desired features from a feature model based on customer requirements and constraints such as feature dependencies. However, the traditional product configuration becomes a time-consuming and error-prone task because of the large number of features and feature relationships. In literature, several approaches have

been proposed to improve the traditional product configuration method. Czarnecki et al. (2005) propose staged configuration which allows incremental configuration of cardinality-based feature model by performing a step-wise specialization of feature models. White et al. (2009) provide automated support of staged configuration based on constraint satisfaction problem (CSP). Mendonca et al. (2008) develop a collaborative product configuration method which decomposes a feature model into several configuration spaces. Each type of stakeholder makes feature selection in a corresponding configuration space and finally the selected features in different configuration spaces are merged to get the final configuration. Loesch et al. (2007) simplify product configuration by reclassifying variable features based on their usage and by restructuring feature model to simplify variabilities. The above proposed approaches have improved product configuration process from different aspects.

In this paper, we propose an approach to improve the efficiency of product configuration. The idea of our approach is that, by taking into account of feature dependencies, it is possible to identify a small set of variation points from a feature model. Selecting variants from this variation point set implies the visiting of all the variation points in the feature model, thus we have reduced the number of variation points to visit during the configuration process. As a result, the number of decisions and rollbacks in the configuration process are significant reduced. We have not found any other works along the same line and we believe our method is an innovative approach.

The remainder of the paper is organized as follows. Section 2 and 3 define the basic concepts and propose our approach. Section 4 discusses the adapted simulated annealing algorithm that is the core of the proposed approach. In Section 5, we use an example to demonstrate how our approach works. In Section 6 and 7 we present experiments results which demonstrate the efficiency of our proposed approach. Section 8 concludes the paper and discusses future works.

2 Feature Model and Product Configuration

Feature Model is a key artifact of the software product line engineering. It tells the commonalities and differences among the member products. As we have already mentioned, various types of relationships among the features have been considered to be included in the feature model, many of them are not precisely defined. Indeed, the relationships among the features are complex and hard to describe.

For example, the "Requires" and "Excludes" relationships are simple and most understandable ones, however, the "Impact" relationship is somehow more complex. Feature A has impact on feature B could mean several things, e.g. the selection of feature A suggests in certain degree of selection of feature B, or the implementation of feature A depends on the implementation of feature B etc. So, the "Impact" relationship is defined less precisely and harder to deal with.

If we only consider the simple relationships, i.e. the relationships we could define precisely, then many mathematical approaches could be involved to improve the efficiency of software engineering process. Mannion (2002) uses propositional logic expressions to detect "void feature model" errors and Zhang et al. (2004) develop a propositional logic-based approach to verify partially customized feature models at any binding time. For detecting dead features and false optional features, Czarneck et al. (2005) transform a feature model into an CSP problem which includes a set of variables and a set of constraints over the variables and then uses CSP solvers to automate the identification process. Trinidad et al. (2008) further develop a CSP-based approach to explain the identified feature model errors. To improve the efficiency of CSP-based approaches, we have developed a constraint propagation-based method to identify and explain dead features and false variable features (Zhang et al. 2011). We can use the above mentioned approaches to obtain a valid feature model by detecting and correcting feature model errors and then perform product configuration in the valid feature model.

As we would like to improve the efficiency of product configuration by applying some mathematical approaches, we need to limit us to the dependencies which can be defined properly. In this paper, we only consider the "Requires" and "Excludes" dependencies. We also like to point out that our approach is extendable to cover other types of dependencies if they are defined accurately and the logical operations involving these dependencies are defined precisely.

When configuring a product, we usually need to go through the feature model and make a configuration decision at each variation point to select variant(s). Usually a depth-first traversal of a feature tree will be employed to make decision at each variation point. Assuming that there are two variation points, VP1 and VP2, we first encounter VP1 during the traversal and select a variant at VP1. If the selected variant at VP1 has "Requires" dependency with a variant at VP2, then the required variant at VP2 has to be selected based on the "Requires" dependency. Thus, we do not need to visit this variant at VP2 later. In this case a selection of variants at one variation point may already cover the selections at other variation points. If those variation points with greater coverage are processed first, obviously, there will be less number of decisions to make, thus the configuration process is more efficient. Another advantage of doing configuration this way is that we can reduce the mistakes made during configuration. For the above mentioned example, assume that we make configuration decision at VP2 first and mistakenly decide not to include this variant in the final product. We will not realize this is a wrong decision until we make configuration decision at VP1. In this case, we have to go back to VP2 again to correct the wrong selection made before. And in a worse situation, the corrections might propagate, thus could be time consuming to fix. Thus, in terms of configuration efficiency and quality, it is better to visit VP1 first in the configuration

The sequence of the variation points following which we make our configuration decisions has significant impact on the efficiency of product configuration. Getting the correct sequence is the key idea of our approach.

3 The Proposed Approach

As discussed in the precede section, the sequence of variation points follow which we select the variants is important for improving the efficiency of product configuration. The sequence of the variation points can be determined based on a parameter which we call $Configuration\ Coverage$. Configuration Coverage (CC) of a variation point refers to what extent a configuration decision made at a variation point covers the configuration decisions of the remainder variation points in a feature model. To improve the efficiency of

configuration process, it is crucial to identify a minimum set of variation points, where the decisions made at this set of variation points cover the decisions to be made at all the variation points in a feature model. This set of variation points will be sorted based on their configuration coverage and we will start making configuration decision at the variation point with the biggest configuration coverage. However, as the feature model could be very complex, and the feature dependency model could be hard to trace, it is not always straightforward to find such a minimum set of variation points for product configuration. We propose to employ some well studied mathematical techniques to help identify the minimum variation point set from a feature model. Before we present the proposed approach, we first define some measurements used in the approach.

As mentioned, each variation point consists of a set of variants and a multiplicity. For a variant v, we define two measurements, one is called the *Positive Coverage* $PC(v)$, another one is called the *Negative Coverage* $NC(v)$. When the variant v is included in a product configuration, the positive coverage of variant v ($PC(v)$) is a set of variable features which will be automatically included or excluded based on their dependent relationships with variant v. Similarly, when the variant v is excluded in a product configuration, the negative coverage of variant v ($NC(v)$) is a set of variable features which will be automatically included or excluded based on the multiplicity constraint between these variants. To work out the positive coverage and negative coverage, we need to examine the dependency relationships among the variants in a feature model. For example, if variant v requires variant w, then we know w is in $PC(v)$, furthermore, if w requires variant u, then u is also in $PC(v)$ since if v is included in the final product, then u will be included as well. If variant t requires variant v, then we know t is in $NC(v)$, since if v is not included in the final product, then t can not be included as well.

For a variation point in a feature model, the multiplication rule restricts the selection of the variants associated with the variation point. For example, a multiplicity of $1..n$ means only up to n variants can be selected in the final product. For all the variants associated with a variation point, we call a subset of variants a *valid selection* if it obeys the multiplicity. The *complement* of a valid selection is the set of variants that are not included in the selection at the variation point. When a certain valid selection has been made at a variation point, the configuration coverage of the selection is the union of all the positive coverage of the variants in the valid selection and all the negative coverage of the variants in the complement of the selection. Below is an example to illustrate how do we calculate these parameters. Note a variation point may have different configuration coverage when different valid selections are made at the variation point.

Included in Fig. 1 is a fraction of a feature model. There are four variants $v1$, $v2$, $v3$ and $v4$ associated with the variation point VP, and the multiplicity restricts the selection, i.e. only up to two variants can be included in the final product. From the dependency relationship Fig. 2, we know that:

$PC(v1) = \{u1, u3\}$, $NC(v1) = \emptyset$, $PC(v2) = \{u2, u4, u7, u8\}$, $NC(v2) = \emptyset$, $PC(v3) = \{u5\}$, $NC(v3) = \{u6\}$, $PC(v4) = \{u4, u8\}$ and $NC(v4) = \{u6\}$.

All possible selections based on the multiplicity at the VP are listed below,

$v1$, $v2$, $v3$, $v4$, $v1 \cup v2$, $v1 \cup v3$, $v1 \cup v4$, $v2 \cup v3$, $v2 \cup v4$, $v3 \cup v4$.

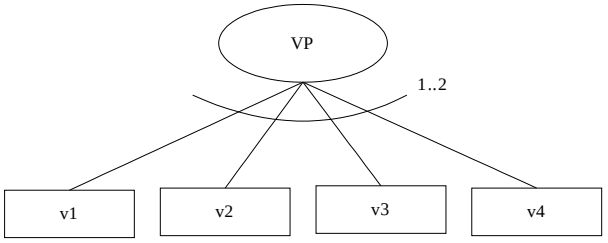

Figure 1: A Variation Point (VP) and its Variants.

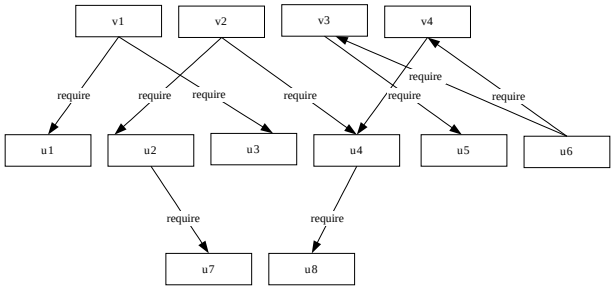

Figure 2: The Dependencies among Variants.

The configuration coverage (CC) of each selection is listed as following:

$$CC(v1) = PC(v1) \cup NC(v2) \cup NC(v3) \cup NC(v4)$$
$$= \{u1, u3, u6\},$$
$$CC(v2) = \{u2, u4, u7, u8, u6\},$$
$$CC(v3) = \{u5, u6\},\ CC(v4) = \{u4, u8, u6\},$$
$$CC(v1 \cup v2) = PC(v1) \cup PC(v2) \cup NC(v3) \cup NC(v4)$$
$$= \{u1, u2, u3, u4, u7, u8, u6\},$$
$$CC(v1 \cup v3) = \{u1, u3, u5, u6\},$$
$$CC(v1 \cup v4) = \{u1, u3, u4, u6\},$$
$$CC(v2 \cup v3) = \{u2, u4, u7, u8, u5, u6\},$$
$$CC(v2 \cup v4) = \{u2, u4, u7, u8, u6\},$$
$$CC(v3 \cup v4) = \{u5, u6\}.$$

The maximum from the above sets is the CC when $v1$ and $v2$ are selected at this variation point, we call this *Maximum Configuration Coverage* ($MAXCC$). The Configuration Coverage of a variation point is the $MAXCC$ of all the valid selections at the variation point. If a selection of variants at a variation point can result in $MAXCC$, we call the selection a *Max Coverage Selection* (MCS). For the above example, $MCS = v1 \cup v2$, $MAXCC = CC$ $(v1 \cup v2) = \{u1, u2, u3, u4, u7, u8, u6\}$.

The $MAXCC$ of a variation point indicates how much a decision made at the variation point covers the decisions at the other variation points of a feature model. The bigger the coverage of a variation point, the (potentially) more variant features will be included/excluded as the result of including or excluding the variation point, therefore, it is more important to visit the variation point earlier in the configuration process. Using the $MAXCC$ of variation points, we can construct a small set of variation points from a feature model, the union of whose $MAXCC$ will cover all the variant features in the feature model. Software engineers should start with this set of variation

points when configuring a product from the feature model. This set of variation points represents the key decisions for configuring a member product. Focusing on this set of variation points will reduce the configuration effort.

One thing we would like to point out is that, in the configuration process, the $MAXCC$ of a variation point might not correspond to the CC of the actual selection. i.e. the actual selection might not give the $MAXCC$. The $MAXCC$ of a variation point only indicates the potential importance of the variation point in the configuration process.

Our approach works like this, we firstly work out the $MAXCC$ of every variation point in the feature model and then calculate the smallest set of variation points that covers the whole feature model. In other words, the union of the $MAXCC$ of this set of variation points includes all the variants of the feature model. The software engineers could start from this set of variation points to configure the final product. Once a decision is made on a variation point (or a set of variation points), we then work out the coverage of the selection(s), this is a straightforward task as we already know the positive coverage and negative coverage of each variable feature. For the rest of the uncovered features in the feature model, we will then repeat this process until all the variants are covered.

To identify a minimum set of variation points which covers a feature model, there are many ways to do it. One of the simple but less optimal approaches would be using greedy algorithm, i.e. selecting the variation points with the biggest coverage until the union of the coverage covers the feature model. A more precise approach is to model the problem as the minimum vertex cover problem and use some approximation algorithm to solve the problem, minimum vertex cover problem is a well studied mathematic problem, there are many approximation algorithms we could use. In the following section, we will discuss this topic.

4 Vertex Cover Problem and Simulated Annealing Algorithm

Before we discuss the vertex cover problem, we firstly describe how do we transfer a feature model into a direct graph. The transformation is quite straightforward, every variable feature in the feature model become a vertex in the resulting graph and the dependencies between two variable features become the arcs in the resulting graph. For example, if feature A requires feature B, then there are two vertices in the resulting graph, say vertex A and vertex B. And also there is an arc goes from vertex A to vertex B in the graph. If feature A excludes feature B, then there will be two arcs between vertex A and B. Once we can model the feature model into a direct graph, we can then apply some discrete optimization techniques.

In graph theory terms, a "vertex-cover" of a directed graph (digraph) is a set of vertices such that each arc of the digraph is incident to at least one vertex of the set. A minimum vertex-cover is a vertex-cover of the smallest size. The problem of finding a minimum vertex-cover is a classical optimization problem in computer science. This problem is a typical example of a NP-hard optimization problem and an optimal solution is very hard to obtain in general. Normally randomized algorithms become the first choice. For example, Simulated Annealing and Genetic Algorithm have been used very often in solving the vertex cover problem. The algorithms are efficient and very often can produce reasonable good solutions.

The problem we are dealing with here is very similar to the classical Vertex Cover problem. The key difference is the coverage in the classical minimum vertex cover problem is the immediate neighbor of the node, where in our problem, the coverage can extend to the nodes beyond the immediate neighbor. However, this difference does not introduce much difficulties into the original problem. We believe that a Simulated Annealing algorithm would still be suitable for solving our problem, as the algorithm can produce reasonable good solutions in a given time frame.

Simulated Annealing (SA) is a well known randomized algorithm in approximating optimal solutions. The technique was proposed by Metropolis et al. (1958) and then been further developed for combinatoric optimization by Pincus (1970).

The technique was originally used as a means of finding the equilibrium configuration of a collection of atoms at a given temperature. Analogy with the physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the solutions and on a global parameter T (called the temperature), that is gradually decreased during the process. The current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. Occasionally, we allow the current solution to go worst, which prevents the method from becoming stuck at local minimum.

The pseudo code of the Simulated Algorithm 4.1 for solving the minimum vertex cover problem is given below. In the inner "for loop", an original vertex-cover is created at random using "Breadth-First Search" (BFS). BFS is a graph searching algorithm that begins at a root vertex (also selected at random base) and explores all the neighboring vertices. Then for each of those neighboring vertices, it explores their unexplored neighbor vertices, and so on, until it exhausts all the vertices. Once the for loop terminates, we have an vertex cover which might not be optimal. Next, we start the SA process, staring from the vertex-cover found before, we shall randomly choosing a vertex from the original vertex-cover to be replaced, in order to obtain a new vertex-cover. If a smaller vertex-cover is produced, then we continue from the new vertex-cover. Otherwise, with certain probability, say $e^{-\Delta/T}$, we still continue from the new vertex-cover, where T is a global time-varying parameter called the *Temperature* and Δ is the increase in cost (i.e., $|VC(G)| - |VC_{min}(G)|$). A limited number of iteration is accepted at each Temperature level. The optimality of the results depends on the number of iterations. The more iterations we run, the results we found are closer to the optimal results, however, the more running time it will consume.

The HSAGA algorithm is an improvement of the standard SA algorithm and was introduced by Tang et al. (2008). The algorithm combines the Genetic Algorithm (GA) and Simulate Annealing (SA) algorithm. The HSAGA algorithm have multiple iterations, in each iteration, we start from multiple instances, i.e, solutions or partial solutions, and we apply SA on these instances to produce the offsprings. The offsprings will then be crossed over in the GA algorithm and producing instances for next iteration. The key idea is to balance the effort in local optimization and multiple probing. The algorithm has demonstrated its efficiency in several classical discrete optimization problems, for more details on the HSAGA algorithm, please see works by Tang et al. (2008).

We have applied the HSAGA algorithms in finding

the minimum cover of the feature model. We have modified the algorithm and the algorithm will start from a random selected variable feature in the feature model, since we know it is coverage, i.e. the set of vertices covered by the vertex, so we can remove the vertex and also the set of vertex it covers, and then we randomly select another vertex and repeat the same process. When this process finishes, we will have a cover of the feature model. We will produce multiple covers in the same way and then apply the HSAGA algorithm on the instances. We would like to note that the HSAGA do not out perform the standard SA algorithm if the instance is small, for example, if the graph has less than a few thousand nodes.

Algorithm
4.1: SIMULATED ANNEALING(G)

comment: $T = 1.0$, $CR = 0.75$, $VC_{min}(G) = \{\}$
while No change in $\mathrm{VC_{min}(G)}$
do
  for $i \leftarrow 0$ **to** Iteration-length
    do
      Generate a VC(G)
      $\Delta \leftarrow |\mathrm{VC(G)}| - |\mathrm{VC_{min}(G)}|$
      if $\Delta < 0$
        then $\mathrm{VC_{min}(G)} \leftarrow \mathrm{VC(G)}$
        else if $random[0;1) < e^{-\Delta/T}$
        then $\mathrm{VC_{min}(G)} \leftarrow \mathrm{VC(G)}$
  T=T*CR;
return ($\mathrm{VC_{min}(G)}$)

- $VC(G)$ - A vertex-cover of a directed graph G.
- $VC_{min}(G)$ - the minimum vertex-cover of a directed graph G.
- T - Temperature, usual the initial value of Temperature is 1.0.
- CR - Cooling-rate, typical values for Cooling-rate are in the range from 0.75 to 0.98.
- *Iteration-length* - A limited number of iteration is accepted at each Temperature level. For better results, we use $100{*}|V(G)|$ as the maximum number of iteration, where $V(G)$ is the total number of vertices in G.

In this paper, we use $100 * |V(G)|$ as the maximum number of iteration. Once the maximum number of iteration has been reached, the temperature is lowered and a new iteration begins. If a more accurate solution is expected, then the number of iteration should be increased.

5 Case Study

In this section, we will introduce a case study to illustrate how our approach works. In Fig. 4, we present a modified version of a Library System feature model based on our previous work (Lin et al. 2010), where 19 variant points are added to make the feature model non-trivial. As we can see, there are over 60 variable features under 42 VPs. Each variation point is represented by a name, such as VP1 and VP2. A hollow circle indicates the variation point that is linked to a set of variants. Features linked to a solid circle in the figure are mandatory features. The dependencies among the variants are presented in Table 1. Each variant listed in the table is assigned with a Variant ID, called VID. The "Requires" and "Excludes" columns represent the dependencies among the variants. We use this dependency information to create a directed graph shown in Fig. 3. In this directed graph each variant is represented as a node labeled by its VID, dependencies among the variants are represented as arcs among them. For example, the "Requires" relationship between the variant "Reward Point" (VID =11) and the variant "Reward Policy" (VID=3) shown in Table 1 corresponds to an arc from the node 11 to 3 in Fig. 3.

Using this directed graph, we calculate the $MAXCC$ for each variation point. Table 2 shows the $MAXCC$ of each variation point and the corresponding MCS. The MCS column in the table means the corresponding selection of variants that results in $MAXCC$ for the variation point. For example, for $VP7$, the MCS is $4 \cup \neg 5$, it means that if we include variant 4 but do not include variant 5, this selection of variants at $VP7$ will result in the inclusion/exclusion of another 4 variants from other variation points as shown in the table. This set of variants is MCS of $VP7$.

Based on the $MAXCC$ of each variation point, we can find a set of variation points which covers the feature model, and the set is the smallest as possible. If the MCS of each variation point in this set is selected the whole feature model will be covered, i.e. we do not need to go through other variation points for the product configuration. If in the configuration process, at a variation point, the selection is not the one giving the $MAXCC$, then we will have to recalculate the covering set.

A minimum covering set for the Library software product line has been identified and sorted into a sequence in terms of the size of their CC which are shown in Table 3. We can see that the variation points $VP35$, $VP9$, $VP2$, $VP8$, $VP11$, $VP10$, $VP30$, $VP15$, $VP6$, $VP27$, $VP34$, $VP31$, $VP40$, $VP41$ and $VP39$ covers all the variants of the feature model. In this particular feature model, some of VPs do not have any dependency relationships with other variants except their own parent or children variants, such as $VP4$, $VP5$, $VP26$, $VP32$ and $VP42$, which are not listed in Table 2 and we will deal with these VPs after we processed the others.

Since $VP35$ has the biggest coverage in the sequence, so software engineers should start the configuration by examining the variants associated with the $VP35$ and making selections. Assuming that the selection of variants at $VP35$ is the same of its MCS, the configuration will continue to select variants from $VP9$ that is the second in the sequence. The configuration process is going to continue until all the variation points in the sequence have been visited, we then get a product configuration. This is an ideal situation where the MCS of each variation in the sequence is selected. However, if the selected variant set at a variation point is not its MCS then the covering set should be recalculated. Suppose that the selection of variants at $VP8$ is 6 that is not MCS of $VP8$, in this case, we will first work out the coverage of the new selection, which is $\{5, 46\}$, and remove this set of variants from the directed graph we have generated at the beginning of this section. Then we will recalculate the cover for the left over variant features. This cover is shown in Table 4.

Using $MAXCC$ of a variation point as its Configuration Coverage works for those large feature models which contain dense dependency relationships. However, for small and simple feature models, we recommend to use the *Average CC* instead. The reason is that, for small and simple feature model, very often the $MAXCC$ of a variation point significantly larger than the CC of the other selections, giving somehow false indication of how important is

Table 1: The Dependency Relationships among Variants in Feature Model.

VID	Variant	Requires	Excludes	VID	Variant	Requires	Excludes
0	Payment			31	Overdue Fee	0,2,12,17	
1	eBook	41,51	48	32	Damage Cost	0,2	
2	Fee Policy			33	Reserve Fee	0,2	
3	Reward Policy			34	Online Reserve	51,57	48
4	Registration Fee	0,2		35	Online Cancel	51,57	48
5	Issue Library Card	46		36	Overdue Notification	12,17	
6	Replace Library Card	5,46		37	Fee Notification	2,12,17	
7	Renew Fee	0,2		38	Email	10	
8	Phone Number			39	Post	9	
9	Address			40	SMS	8	
10	Email Address			41	Digital Library	51,53	48
11	Reward Point	3,12		42	On-site Explore	51,57	48
12	Borrowing History			43	Web Explore	51,57	48
13	Update Phone Number	8		44	On-site Print	50	48
14	Update Address	9		45	Download	51,57	48
15	Update Email Address	10		46	Library Card Device		
16	Loan Fee	0,2		47	Self-check Device		
17	Record Check	12		48	Non-Network		1,22,24,25,26, 28,29,34,35,41,42, 43,44,45,49,57
18	Pops-up Reminder	12		49	Network Based		48
19	Loan Restriction	12,17		50	LAN Based		
20	Front Desk	12,17,18		51	Internet Based	53	
21	Self-check	12,17,18,47		52	Wireless Device		
22	Web Search	51,57	48	53	Network Security		
23	View Account	12		54	Message Encryption		
24	Website	51,57	48	55	Use Library Card	5,46	
25	On-site Computer	50	48	56	Use Digital Certificate	59	
26	Inter-Library		48	57	User Web Interface	51,53	48
27	Onsite Loan			58	Credit Card	54	
28	Web Request	51,57	48	59	Digital Device		
29	InterLibrary Search	56	30,48	60	Firewall		
30	External Database		29	61	Proxy Server		

Table 2: Max Configuration Coverage of Each VP.

VPID	MCS	MAXCC	VPID	MCS	MAXCC
VP1	¬0	{4,7,16,31,32,33}	VP22	35	{48,51,57}
VP2	¬2 ∪ ¬3	{4,7,11,16,31,32,33,37}	VP23	36 ∪ 37	{2,12,17}
VP3	1	{41,48,51}	VP24	38 ∪ 39 ∪ 40	{8,9,10}
VP6	11	{3,12}	VP25	23	{12}
VP7	4 ∪ ¬5	{0,2,6,55}	VP27	24 ∪ 25	{48,50,51,57}
VP8	6 ∪ 7	{0,2,5,46}	VP28	26	{48}
VP9	¬12	{11,17,18,19,20, 21,23,31,36,37}	VP29	28	{48,51,57}
VP10	¬8 ∪ ¬9 ∪ ¬10	{13,14,15,38,39,40}	VP30	29	{30,48,56}
VP11	13 ∪ 14 ∪ 15	{8,9,10}	VP31	41	{48,51,53}
VP12	16	{0,2}	VP33	42 ∪ 43 ∪ 44 ∪ 45	{48,50,51,57}
VP13	¬17	{19,20,21,31,36,37}	VP34	¬46 ∪ ¬47 ∪ ¬52	{5,6,21,52,55}
VP14	19	{12,17}	VP35	48	{1,22,24,25,26,28,29,34, 35,41,42,43,44,45,49,57}
VP15	21	{12,17,18,47}	VP36	50 ∪ ¬51	{1,22,24,28,34,35, 41,42,43,45,57}
VP16	¬18	{20,21}	VP37	¬57	{22,24,28,34,35,42,43,45}
VP17	22	{48,51,57}	VP38	¬53	{41,51,57}
VP18	31	{0,2,12,17}	VP39	¬54	{58}
VP19	32	{0,2}	VP40	55 ∪ 56	{5,46,59}
VP20	33	{0,2}	VP41	58	{54}
VP21	34	{48,51,57}			

Table 3: A Sequence of VPs which covers Feature Model.

VP	MCS	MAXCC
VP35	48	{1,22,24,25,26,28,29,34, 35,41,42,43,44,45,49,57}
VP9	¬12	{11,17,18,19,20,21,23,31,36,37}
VP2	¬2 ∪ ¬3	{4,7,11,16,31,32,33,37}
VP8	6 ∪ 7	{0,2,5,46}
VP11	13 ∪ 14 ∪ 15	{8,9,10}
VP10	¬8 ∪ ¬9 ∪ ¬10	{13,14,15,38,39,40}
VP30	29	{30,48,56}
VP15	21	{12,17,18,47}
VP6	11	{3,12}
VP27	24 ∪ 25	{48,50,51,57}
VP34	¬46 ∪ ¬47 ∪ ¬52	{5,6,21,52,55}
VP31	41	{48,51,53}
VP40	55 ∪ 56	{5,46,59}
VP41	58	{54}
VP39	¬54	{58}

the variation point to the configuration process. So the *Average CC* gives better guide in the process. *Average CC* is defined as the set which includes the variants that appear more than once in the *CC* of all the valid selections at the variation point. For example, there are four possible selections at $VP2$, $\neg 2 \cup \neg 3 = \{4, 7, 11, 16, 31, 32, 33, 37\}$, $2 \cup 3 = \emptyset$, $\neg 2 \cup 3 = \{4, 7, 16, 31, 32, 33, 37\}$, $2 \cup \neg 3 = \{11\}$. So the average CC is $\{4, 7, 16, 31, 32, 33, 37\}$.

We have tested our approach on this specific library system feature model. We also conducted experiments on several random generated feature models. All experiments generated good results on corresponding feature models. According to the results, our approach reduces the configuration efforts spent on feature decisions significantly. In next two sections, we will introduce the processes of our experiments and explain the related results.

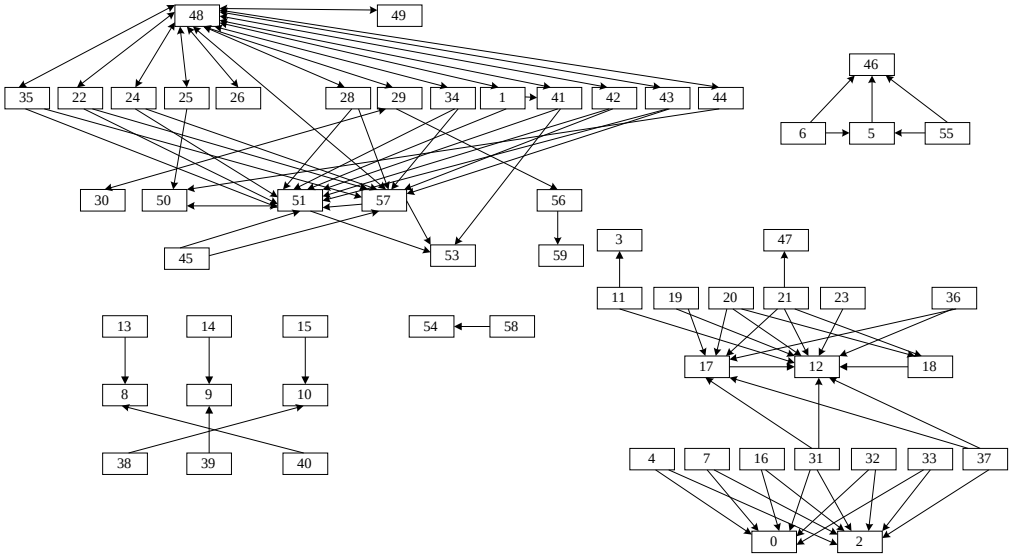

Figure 3: Dependencies among Variants in Library Systems.

Table 4: A Sequence of VPs after Decision made at VP8.

VP	MCS	MAXCC
VP35	48	{1,22,24,25,26,28,29,34, 35,41,42,43,44,45,49,57}
VP9	¬12	{11,17,18,19,20,21,23,31,36,37}
VP2	¬2 ∪ ¬3	{4,7,11,16,31,32,33,37}
VP11	13 ∪ 14 ∪ 15	{8,9,10}
VP10	¬8 ∪ ¬9 ∪ ¬10	{13,14,15,38,39,40}
VP30	29	{30,48,56}
VP15	21	{12,17,18,47}
VP6	11	{3,12}
VP27	24 ∪ 25	{48,50,51,57}
VP34	¬46 ∪ ¬47 ∪ ¬52	{5,6,21,52,55}
VP31	41	{48,51,53}
VP40	55 ∪ 56	{5,46,59}
VP12	16	{0,2}
VP41	58	{54}
VP39	¬54	{58}

6 Experiment Results

We had conducted an experiment by using the Library System feature model. Firstly, we developed the requirement documents for two library systems based on the feature model. One with our university library in mind, the library is not only available for students and staff, but also available for public access. The other library system is a medium sized community library for local residents. The university library is more complex than the community library, for example, the digital library access is part of the university library but not for the community library.

The requirement documents of two systems serve two purposes, 1) to give users some general ideas of what features the library system might include, 2) to control the scope of the final product. Because the final products are configured using the same feature model, we presume that, for a given library requirement, the products will be similar, i.e. with minor differences in terms of features included. If one of the final products been configured in the experiment is significantly different from other products, for instance, contains a lot more features than the average case, then it is not a valid sample and we will reject the product. The reason to do so is to have some consistency among the configured products so we could compare the result. Furthermore, as the economical concerns, such as software development cost, are not included in the feature model, the configured product could be unrealistic and having too many or too little features, thus we have reject these products in our experiment. To better simulate the real world situation, where the requirements are very often not clear and exact, some vague descriptions and even misleading information are introduced to the requirement documents.

The experiment consisted of two focus groups with 20 university students. Users were picked randomly without SPL experience to make sure they are on the same level of background. The basic idea of SPL and purpose of experiment were explained to all the users in two groups. One group used traditional configuration method (depth-first traversal to get through each variant) and the other group used our approach based on variant points sequence generated from *Average CC*. The experiment was carried out in pairs, i.e. one user from each group. Based on the functional and non-functional requirements listed in the requirement document, users made their selections on the feature model. Each user is given a computer program that could record all the decisions he/she have made, when the computer program identified the conflicts among the selected features, the computer program will remind the experiment conductor who then explains the conflict and help user to change their selections. If there was anything user feels unclear about, then he/she could ask questions during the experiments, the information was shared with the peer in the experiments. This parallel process was able to provide a direct comparison of two methods. For each group, 50% of users configured the university library system and the other half configured the community library.

Several key data sets were collected during experiment process. Including the number of Decisions (users' selection of variants), the number of Rollbacks (users has realized that they had made a mistake and modified their previous selection) and Time con-

sumed for configuration. The number of Decisions would be the most important figure to demonstrate the efficiency of our approach.

Considering the variant features involved in the dependency relationships, they form trees/chains. In our approach, configuration decision is always made at the root of the tree/chain, thus, less number of Rollbacks. Rollback represents errors in the configuration, where the impact of one improper selection does not reflect immediately but afterwards. It seems little happened in our approach. Time is a parameter to support configuration procedure somehow. It only reflects the process to some extent because there are so many factors may influence the time consumed. For instance, users' understanding of the features.

Table 5 shows critical parameters collected for randomly selected 10 pairs of users in our experiment. Data for each pair is included in a row in Table 5 for comparison. Rollbacks did not happen at all for the user using our approach. Time consumed by users using our approach is much less than group using traditional approach when configuring the same product. From the "Decision" column of two groups, we can see the gaps between this two approaches are obvious even for this simple feature model. Figures in this experiment may vary slightly but we can still see the tendency. Our approach definitely has advantage in terms of the number of decision to make and the number of Rollbacks. Table 6 shows average value comparison of two approaches.

Table 5: Tradition Configure Group(left) and Optimization Configure Group(right).

User No.	Decision	Rollback	Time(mins)	User No.	Decision	Rollback	Time(mins)
1	42	0	30	2	34	0	23
3	43	1	25	4	34	0	21
5	42	0	23	6	34	0	20
7	42	0	28	8	33	0	22
9	43	1	30	10	34	0	26
11	42	0	28	12	32	0	24
13	42	0	24	14	33	0	16
15	42	1	20	16	33	0	15
17	44	2	28	18	32	0	22
19	42	0	30	20	33	0	24

Table 6: Average Figures Comparison.

Group	Average steps	Average time	Average decision reduced(%)
Traditional	42.4	26.6	21.7%
Optimization	33.2	21.3	

7 Experiment on Random graphs

In the literature (Segura et al. 2010), there are some works using random generated feature model as the testing environment. Here we have also conducted an experiment on random generated feature model. The random feature model is generated in three steps, first a random graphs is generated based on the simple random graph model by Erdos and Renyi (Erdos et al. 1959), where we start with a fixed set of vertices and add edges to the graph based on a edge probability parameter. In here, a vertex represents a variable feature, an edge represents a dependency relationship in a graph. In the second step, we randomly generated the relationships between the features. Except "Requires" and "Excludes" relationships, we also included parent-children relationships and the multiplicity constraints between variants in the random feature models. Of cause, it is easy to see that here, very likely, large number of conflicts will be generated, so we have to run through the third step, where we go through all the cycles in the generated graphs to check for inconsistency. As explained before, a cycle possibly represents a conflict, therefore, we have to randomly remove an edge a cycle to destroy the cycle, thus to remove the conflict. Once we complete these three steps, we then have a valid feature model for configuration.

The configuration process is automated as well, a computer program randomly selects a feature and then check if the selection conflicts with previous decisions, if yes, then the program will make a different selection or randomly change the previous decisions to remove the conflicts. The program repeats the process until all the features are visited. When using our proposed approach, there will be no rollbacks.

Obviously, the final products generated by the random configuration can be quite different. i.e. a product could contain large number of features while the others might contain significantly less number of features. To maintain the similarity of the final products, we reject those products (i.e. invalid products) which contain more than 85% of features or less than 50% of features of the feature model.

We have conducted our experiments on a series of random systems. We have generated three random systems of 800 nodes, 2000 nodes and 3000 nodes of different edge probabilities. We then configure 1000 valid products using each of the configuration methods on the three random systems. And we include the exact numbers of the decisions of our approach/the traditional approach in Table 7 below.

Table 7: Experiment Results of Random Systems.

Edge Probability	Random graph with 800 nodes	Random graph with 2000 nodes	Random graph with 3000 nodes
0.01	264/413	330/473	361/498
0.02	219/307	255/374	298/411
0.1	124/169	145/186	167/199
0.2	91/112	122/157	133/145

The result is displayed in Table 7. Columns represent three random systems of 800, 2000 and 3000 nodes. Rows indicate edge probability when creating the random graph and corresponding cells represent the exact decision numbers of both approaches (i.e our approach vs traditional). With the growth of the edge probability, we can see that the decisions made between two approaches becoming closer. This is because that the number of edges is growing, and the dependencies relationships among the variation points become dense, thus more likely the random selection has large coverage, and the gaps decrease. From Table 7, it is quite clear that using our approach is more efficient in the product configuration.

8 Conclusions and Future Works

In this paper, we have presented an approach to improve the efficiency of product configuration in software product line. Using this approach, a set of variant points (VPs) is identified from the feature model. This set only contains small number of VPs but the union of CC of these VPs will cover all the variants of the feature model. To configure a product, instead of making configuration decision at every VP of the feature model, we only go through small number of VPs

to make configuration decisions. In this way, the number of decisions is reduced, thus the effort of decision making is saved. Furthermore, using this approach, it is less likely to make mistakes in the configuration, as the we have already incorporated the dependency relationships among the variants in our approach.

However, as we have pointed out, that our approach only considers two simple relationships among the variable features, therefore, how to extend our approach to cover other types of relationships is a challenge and we would like to consider this in our future works. Meanwhile, we would like to perform our experiments on some publicly available and complex feature models to further evaluate our approach, we believe that the results must be interesting to many researchers.

References

Benavides, D., Segura, S. & Ruiz-Cortes, A. (2010), Automated Anslysis of Feature Models 20 Years Later: A Literature Review, *Information Systems*, 35(6), 625–636.

Czarnecki, K., Helsen, S. & Eisenecker, U. (2005), Formalizing Cardinality-based Feature Models and Their Staged Configuration, *Software Process: Improvement and Practice*, 10(1), 7–29.

Czarneck, K. & Kim, P. (2005), Cardinality-basd Feature Modeling and Constraints: A Progress Report, *in* 'International Workshop on Software Factories at OOPSLA 2005', San Diego, California, USA.

Erdos, P. & Renyi, A. (1959), On Random Graphs, *Publicationes Mathematicae*, 6, 290–297.

Kang, K., Cohen, S., Hess, J., Nowak, W. & Peterson, S. (1990), Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, Software Engineering Institute, Carnegie Mellon University.

Lee, K., Kang, K. & Lee, J. (2002), Concepts and Guidelines of Feature Modeling for Product Line Software Engineering, *in* '7th International Conference on Software Reuse: Methods, Techniques, and Tools', pp. 62–77.

Lin, Y. Q., Ye, H. L. & Tang, T. M. (2010), An Approach to Efficient Product Configuration in Software Product Lines, *in* '14th International Conference on Software Product Lines', Jeju Island, South Korea, pp. 435–439.

Loesch, F. & Ploedereder, E. (2007), Optimization of Variability in Software Produce Lines, *in* '11th International Software Product Line Conference', pp. 151–162.

Mannion, M. (2002), Using First-Order Logic for Product Line Model Validation, *in* '2nd International Conference on Software Product Lines', pp. 149–202.

Mendonca, M. Cowan, D. Malyk, W. & Oliveira, T. (2008), Collaborative Product Configuration: Formalization and Efficient Algorithms for Dependency Analysis, *Journal of Software*, 3(2), 69–82.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. & Teller, E. (1958), Equations of State Calculations by Fast Computing Machines, *Journal of Chemical Physics*, 21, pp. 1087–1092.

Pincus, M. (1970), A Monte Carlo Mehtod for the Approximate Solution of Certian Types of Constrained Optimization Problems, *Operations Research*, 18(6), 1225–1228.

Pohl, K., Böckle, G. & van der Linden, F. (2005), Software Product Line Engineering: Foudations, Principles, and Techniques, *Springer*, Heidelberg, (2005).

Segura, S., Hierons, R. M., Benavides, D. & Ruiz-Cortes, A. (2010), Automated Test Data Gneration on the Analyses of Feature Models: A Metamorphic Testing Approach, *in* '3rd International Conference on Software Testing, Verification and Validation', Paris, France, pp. 35–44.

Tang, J. M., Miller, M. & Lin, Y. Q. (2008), HSAGA and its application for the construction of near-Moore digraphs, *Journal of Discrete Algorithms*, 6(1), 73–84.

Trinidad, P., Benavides, D., Duran, A., Ruiz-Cortes, A. & Toro, M. (2008), Automated Error Analysis for Agilization of Feature Modeling, *Jouranl of Systems and Software*, 81(6), 883–896.

White, J., Dougherty, B., Schmide, D. C. & Benavides, D. (2009), Automated Reasoning for Multi-step Feature Model Configuration Problems, *in* '13th International Software Product Line Conference', San Francisco, USA, pp. 11–20.

Ye, H. L. & Zhang, W. (2008), Formal Definition of Feature Models to Support Software Product Line Evolutions, *in* '2008 International Conference on Software Engineering Research Practice', Las Vegas, Nevada, pp. 349–355.

Zhang, W., Zhao, H. Y. & Mei, H. (2004), A Propositional Logic-based Method for Verification of Feature Models, *Formal Methods and Software Engineering*, 3308, 115–130.

Zhang, G. H., Ye, H. L. & Lin, Y. Q. (2011), Feature Model Validation: A Constraint Propoagation-based Approach, *in* '10th International Conference on Software Engineering Reaserch and Practice', Las Vegas, Nevada.

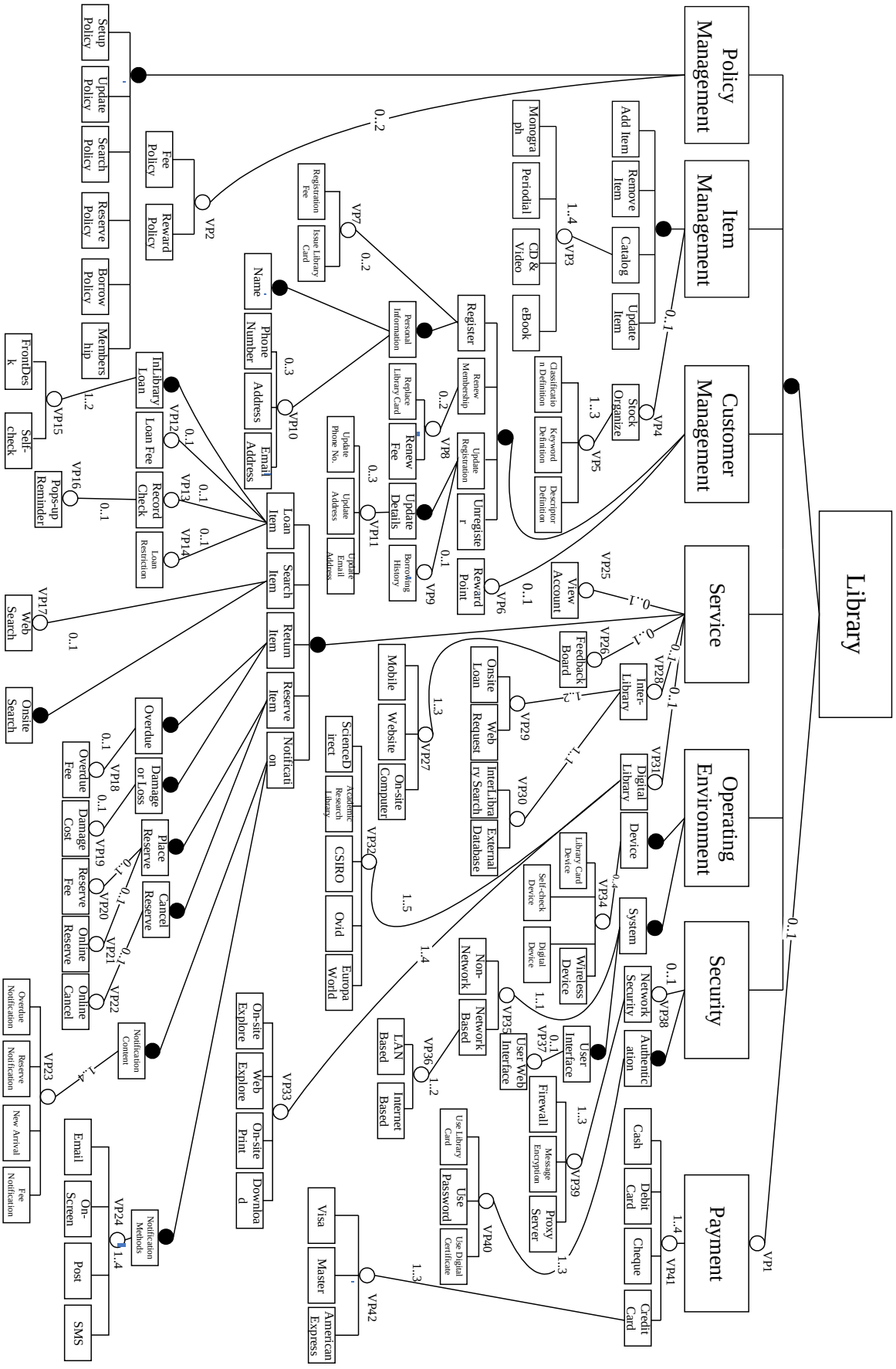

Figure 4: A Feature Model for SPL of Library Systems.